
# University of Limerick
# Department of Sociology Working Paper Series

*Working Paper WP2013-01*
*March 2013*

Brendan Halpin

Department of Sociology

University of Limerick

## *Imputing Sequence Data: Extensions to initial and terminal gaps, Stata's* `mi`

Available at `http://www.ul.ie/sociology/pubs/wp2013-01.pdf`

# Imputing Sequence Data: Extensions to initial and terminal gaps, Stata's `mi`*

Brendan Halpin, University of Limerick

Mar 6 2013

## Contents



## 1 Introduction

In a previous paper (Halpin, 2012) (downloadable) I have described a strategy for multiple imputation for missing data in sequence data (that is, in categorical time series), where missingness tends to be consecutive and take the form of gaps. In this note I document two improvements made to the initial approach: the use of Stata's built-in `mi` multiple-imputation framework for prediction, and the imputation of gaps at the beginning and end of the sequence (which was allowed for, but not implemented, in the original version). I also document the implementation of the algorithm in Stata, in more detail than previously (see section Code).

*Version: update.org,v 1.4 2013/03/10 15:30:42 brendan Exp

The key innovation of the general approach is to handle the sequences or time-series in "long" format (one observation per person per time-unit), and to impute particular observations using lags and leads. The order of imputation is carefully calculated to produce longitudinal consistency. In contrast, multiple imputation by chained equations normally imputes all missings in a given variable, using other variables from the same observation.

The previously documented version imputed only internal gaps. The extension to imputed initial and terminal gaps substantially increases the number of whole sequences available for analysis, and the use of core Stata commands makes the imputation more standard and reliable (but also in the current implementation, distinctly slower: see section Parallel computing for a way of mitigating this).

## 2 Stata's `mi` framework

In the original version, imputations are based on simple predictions from a multinomial logistic regression, where each imputation is drawn at random according to the predicted probabilities of the states. This is not standard multiple imputation practice, where the imputations are normally created by drawing parameter estimates at random from their distribution (i.e. $\sim N(\hat{\beta}, \sigma_{\hat{\beta}})$), calculating predicted probabilities from those, and then drawing at random from the predicted probabilities. This is a more delicate operation, and since it is handled in a robust fashion by Stata's `mi` framework it is as well to exploit that. Additionally `mi`, on the example of the earlier user-written `mice` framework, uses "augmented" logistic regression where "perfect prediction" is encountered. Perfect prediction occurs where, for particular configurations of explanatory variables, there is no variation in the observed state, and thus parameter estimates tend to infinity. Stata usually deals with this problem by dropping the relevant cases (with a warning). Augmented (binomial or multinomial) logistic regression provides a valid prediction for such cases.

The present approach has a very particular structure that exploits the strengths and constraints of the time-series structure. Rather than impute all missings for a given variable, and then going on to use that variable and others to impute in a further variable, we are imputing particular observations in a single state variable that has multiple, time-indexed, observations per individual. The imputed observations are indexed by where they fall in the missingness run or gap, rather than by where they fall in the overall sequence. The procedure then goes on to impute other observations in the same variable (which fall elsewhere in the gap). This makes it difficult to fit into the overall `mi` framework for handling chained imputations, so our use of the built-in functionality is restricted to calling the core imputation code repeatedly to create a single imputation, and accepting its values for the relevant observations. Compared with the initial version of the algorithm documented in the previous paper (Halpin, 2012), where the prediction model was fitted once and used repeatedly, and with standard `mi` where multiple imputations are drawn at each pass, this is very slow (see below for a discussion of ameliorating this by use of multiple parallel Stata jobs, exploiting multi-processor computers without needing StataMP). It is possible that the standard `mi` framework permits a more efficient solution, or that the imputation code can be adapted to save

the prediction model on the first run and re-use it for subsequent runs (in the manner the original version of this code saved and re-used model estimates); these possibilities are being investigated.

For the moment, while the approach is inefficient, it has the benefit of using standard robust code to do the imputations, code which furthermore solves problems such as perfect prediction.

# 3 Initial and terminal gaps

The original version of this procedure addressed only internal gaps, that is, runs of one or more consecutive missing values for which there was at least some prior and subsequent information. It was noted that extension to initial and terminal gaps should be straightforward, with the proviso that the length-limit for an imputable gap should be shorter for an edge gap than an internal one. The structure of the imputation is also a little simpler, in that while for an internal gap it is desirable that the imputation sequence alternates from end to end, for an edge gap the process can go in one direction (starting with the missing case nearest the observed data and moving away). Furthermore, given that there is only prior *or* subsequent information, the prediction model is of necessity a little simpler. When imputing an element of a internal gap, the prediction model uses the last known (or imputed) state, and the next known state, where "state" can mean the value of the state variable but also other information like cumulated duration (prior or subsequent) or other information known for that time point. The edge-gap model has half of this information, and in my example I limit its operation to edge gaps of length up to half the limit for internal gaps (the intuition here is that the maximum distance to observed data will be the same for both sorts of gaps).

Imputing edge gaps would be easier (than internal gaps) to handle in a conventional chained-equation multiple-imputation framework (with the data in wide rather than long format) as the sequence of operations is simpler than for internal gaps. For instance, for initial gaps up to length 5, we first impute element 5 using information from element 6 and later, then element 4 additionally using information (known and imputed) for element 5, and so on. Moreover, for edge gaps the pattern of missingness is by definition monotone (while the pattern of missingness across the whole data set for elements 1 to 5 may not be monotone, the non-monotone missingness will be due to internal gaps that are near the edge). There is hence an argument for imputing edge gaps in "wide" format using the conventional `mi` chained-equation approach (however, it might be difficult or impossible to fit closely equivalent predictive models, particularly in terms of predicting $x_t$ by $x_{t-1}$) but for the moment I present an approach as close as possible to the internal-gap algorithm.

# 4 Imputation sequence

The sequence of operations for internal gaps is described in the previous paper. Essentially we begin with the, say, last element of the longest gap and impute that, taking account of information available immediately afterwards (a lead of 1) and before the start of the gap (a lag of the gap-length). Now the longest

gap is one shorter, and we impute the first element of that gap, using a lag of 1 and a lead of the new gap-length. Note that for some cases, the lead data will now be imputed. In the current version the imputation model is fitted again for each imputation, though it is almost surely sufficient to fit it once for each element and predict repeatedly from the same model. This is because the additional information we can bring in to subsequent estimations due to earlier imputations is quite slight. In conventional multiple imputation by chained equations, imputing missings in a variable allows the inclusion of all that variable's data in subsequent imputation models, but in this framework we can already use nearly all observed data in the time-series state variable, and each imputed observation brings in perhaps a single datum.

In the example implementation discussed in the previous paper, the imputation model uses the prior (lag) and subsequent (lead) state, interacted with a quadratic time-index, plus the interaction of prior and subsequent state with an observation-type variable (see the previous paper; this uses structural information about the data collection which has a bearing on the probability of a spell beginning or terminating on this date). It additionally uses information on the proportion of time before and after spent in each state to allow "history" to have an impact (since the concern is with the joint distribution of the past, present and future states, rather than modelling a theoretical mechanism, it is appropriate to consider the future as having a statistical impact on the present). In Stata terms the model is as follows:

```
mlogit state i.next##c.t##c.t i.last##c.t##c.t ///
             before1 before2 before3 ///
             after1 after2 after3 ///
             i.on##i.n2 i.ol##i.l2
```

(Note that `///` at the end of a line allows Stata commands to take more than one line.) The variables `last` and `next` are prior and subsequent state, `t` is a time-unit index (from 1 to 73 in this case), `ol` and `on` are lags and leads of the `obstype` variable, and `l2` and `n2` are collapsed versions of the 4-category prior and subsequent state variables (collapsed due to sparseness in interaction with observation-type). This model puts a lot of weight on the matrix of transition rates, but allows its influence to change in a non-linear way along the time-axis. The basic assumption of multiple imputation, that data are missing at random given the model, is required here; the `obstype` variable is one way that relevant parallel information can be brought in, but in general time-dependent values of other variables could be used as well, if appropriate.

Because the model needs to fit over a range of gap-lengths, it is useful to have a fall-back strategy in case of convergence difficulties. For instance, sometimes for shorter gaps there is very little difference between the `last` and `next` variables. My approach is to limit estimation to 40 iterations and if the model hasn't converged to fit this simpler model:

```
mlogit state i.next i.last
```

This is a drastically simpler model, but it contains the two most important predictor variables, and maintains longitudinal continuity. As long as its use is triggered fairly rarely, it should have little detrimental effect on the imputations.

For the edge gaps, the imputation model is simpler. For initial gaps where the next (lead) state is known:

```
mlogit state i.next##c.t##c.t ///
             after1 after2 after3 ///
             i.obstype##i.n2
```

And for terminal gaps:

```
mlogit state i.last##c.t##c.t ///
             before1 before2 before3 ///
             i.obstype##i.l2
```

Effectively, the model used is the relevant half of the internal-gap model. The other key difference is that the process iterates simply (no need for alternation) from the internal end of the gap to the end or start of the time-series.

# 5 Parallel computing

As a result of the way in which the Stata `mi` module is used, the procedure is now much slower. While it is likely that more efficient ways can be found to carry out the integration of `mi`, we can for the moment work around this difficulty by taking advantage of the parallelisability of the algorithm. The multi-processor version of Stata, StataMP, will do this automatically, and will certainly be faster than single processor versions. However, StataMP is expensive and many users have single-processor versions. The task itself is very easy to parallelise: every imputation-sequence happens independently of all others, and can be run simultaneously, on machines with multiple CPU cores or on separate machines. Once the data is prepared for imputation, we can run multiple separate jobs carrying out the imputation sequences. As long as each job is fed a distinct random number generator seed, it will produce distinct imputations. Thus, on a 4-core machine, one can run parallel jobs creating 5 imputations each, and recombine them to have 20 imputations. Jobs can be run in parallel either by opening multiple Stata instances and running the do-files, or by using batch mode (in Windows from separate command windows, in Unix by backgrounding the jobs).

For instance, if the file `bsmi_it.do` is set up to take three parameters (seed, number of imputations, and number of imputations in previous files) we can run four sessions generating three imputations each, by running these four commands in four parallel Stata sessions:

Session 1:

```
. do bsmi_it 123451 3 0
```

Session 2:

```
. do bsmi_it 123452 3 3
```

Session 3:

```
. do bsmi_it 123453 3 6
```

Session 4:

```
. do bsmi_it 123454 3 9
```

The actual value of the seed parameter is unimportant: if it is the same the results will be the same (and thus reproducible), if different, different. The Stata do-file will accept the parameters if the first line takes the form:

```
args seed nseq seqindex
```

The parameters will then be accessible as local macros:

```
set seed `seed'
...
forvalues i = 1/`nseq' {
  [...do the imputations...]
}
...
save imputations_`seqindex', replace
```

Thus the fourth session will run 3 imputations (numbered 10, 11 and 12) and save the file `imputations_9.dta`.

We can also do this in batch. Stata's batch mode will save the output from running `filename.do` in `filename.log`, so it is necessary to make copies of the `bsmi_it.do` file to run multiple instances simultaneously. Given copies as `bsmi_it_a.do` and so on, in Unix the following commands will set four sessions running simultaneously in the background:

```
$ stata -b bsmi_it_a 123451 3 0 &
$ stata -b bsmi_it_b 123452 3 3 &
$ stata -b bsmi_it_c 123453 3 6 &
$ stata -b bsmi_it_d 123454 3 9 &
```

The `$` character represents the Unix prompt, and the `&` character makes the commands run in the background. In Windows something similar can be done in command windows, though as far as I know there is no equivalent of the `&` character. Instead, open multiple command windows (press Windows key and enter `cmd` in the search box, or press `r`, then `cmd`, then return) and run commands like:

```
> stata /b bsmi_it_a 123451 3 0
```

where > represents the Windows command prompt. It is assumed the `stata` program is in the path. See the Stata FAQ on the topic for more detail.

# 6 Code

This section walks through the Stata code used to carry out the example imputations. This is example code, not production quality, and has not been written in a very general manner. However, it should be reasonably easy to adapt by hand for other data sets.

Important differences to handle will include the names of the state variable, and any ancilliary variables such as `obstype`, the appropriate predictive models to fit, and to account for a state variable that has more or less than four values.

To run the example, which uses 73 months of labour market history of women who have a birth in month 25, there are three files, `bsmiprep.do` which sets up the data, `bsmi_it.do` which runs the imputations (if in parallel, as copies `bsmi_it_a.do` etc.), and `bsmicombit.do` which unites the parallel imputations.

Copies of these files are available at http://teaching.sociology.ul.ie/seqanal/mi/.

## 6.1 Preparing the data, `bsmiprep.do`

The file `bsmiprep.do` takes the basic data and processes it for the imputations, producing a set of variables that will be used by the imputation programs.

Begin by loading the data from `bslong.dta` which contains 73 observations per person, with variables `pid`, `t`, `state` and `obstype`.

- `pid` is the person identifier, `t` the time index (1 to 73)
- `state` is employment state (4 values and system-missing)
- `obstype` is a categorical variable marking where the data collection structure suggests that the current month is particularly likely to be the start or end of a spell:

```
use pid t state obstype using bslong
```

Save a copy of the unimputed state variable to compare later:

```
gen oldstate = state
```

Calculate the spell sequence number within individuals. A "spell" is a consecutive run of the same state within the individual trajectory:

```
gen spellno = 1
sort pid t
by pid: replace spellno = spellno[_n-1] + (state!=state[_n-1]) if _n>1
sort pid spellno t
```

Total number of spells per individual:

```
by pid: egen nspells = max(spellno)
```

Calculate gap length (note we keep sequences with gaps longer than the maximum because they contribute to the prediction model):

```
gen lg = 0
by pid spellno: replace lg = _N if missing(state)
```

Index months within spells: this is needed to identify which missing months to impute:

```
gen tw=1
by pid spellno: replace tw = tw[_n-1] + (state==state[_n-1]) if _n>1
```

Identify gaps at beginning or end of the sequence, calculate their length:

```
by pid: gen initgap = missing(state[1])
by pid: gen termgap = missing(state[_N])

gen igapspell = initgap & spellno==1
gen tgapspell = termgap & spellno==nspells

gen igl = 0
by pid spellno: replace igl = _N if igapspell
gen tgl = 0
by pid spellno: replace tgl = _N if tgapspell
```

Calculate prior and subsequent cumulated duration in each of the four states, first as a count of months, then as a proportion of observed time. Note that where state is missing the variable has valid values depending on what went before or came after.

```
sort pid t
forvalues i=1/4 {
  by pid: gen tb`i'=state[1]==`i'
  by pid: replace tb`i' = tb`i'[_n-1] + (state==`i') if _n>1
}
// Map between a count of months to a proportion of time
forvalues i=1/4 {
  by pid: gen before`i' = tb`i'/(tb1+tb2+tb3+tb4)
}

// reverse-sort to look to the future
gsort pid -t

forvalues i=1/4 {
  by pid: gen ta`i'=state[1]==`i'
  by pid: replace ta`i' = ta`i'[_n-1] + (state==`i') if _n>1
}

forvalues i=1/4 {
  by pid: gen after`i' = ta`i'/(ta1+ta2+ta3+ta4)
}

sort pid t
```

Save this data as we will use it repeatedly:

```
save bsmiprep, replace
```

## 6.2 Carrying out the imputations, `bsmi_it.do`

We invoke `bsmi_it.do` (or a copy of it) with command line parameters to set the seed, the number of imputations to carry out, and the number of imputations already carried out by earlier runs (or simultaneous, parallel runs, see above). For instance, from the Stata command-line:

```
. do bsmi_it 123455 4 8
```

will set the seed of the random number generator to 123455, and create four imputations, numbering them from 9 to 12.

The code follows, with the first line accepting the command line parameters as local macros, and the subsequent lines setting up some global macros

```
args seed maxreps repoffset

set seed `seed'

global longestgap   12   // Longest internal gap to impute
global longestgapit  6   // Longest initial/terminal gap to impute
global maxreps `maxreps' // How many imputations to do
```

At this point in the file, two "programs" are defined, but here I leave their presentation for later (see below) and examine the main loop which carries out the number of imputations given by the second command line parameter. It begins by loading the prepared data file, determining the length of the sequences (storing it in the global macro `$totlen`), and initialising two variables that will be used and changed.

```
forvalues iter = 1/$maxreps {

  use bsmiprep, clear

  qui by pid: su t
  global totlen `r(max)'

  qui gen last = .
  qui gen next = .
```

Then, with a loop within the main loop, it repeatedly calls the internal-gap imputation program (as many times as the maximum permitted gap length):

```
forvalues i = 1/$longestgap {
  sort pid t // needed beccause MI affects sort status (but not sort order)
  do_imputations `i'
}
```

Analogously, for as many times as the longest initial or terminal gap permitted, the edge-imputation program is called:

```
forvalues i = 1/$longestgapit {
  sort pid t
```

```
  do_imputations_it "i" `i'
  sort pid t
  do_imputations_it "t" `i'
}
```

Repeatedly calling these imputation programs will have created a single complete imputation (apart from gaps whose length is greater than permitted).

To finish the loop, an iteration number is created, which is consistent across the multiple runs of `bsmi_it.do`; all but the essential variables are dropped, and the long-format data set is reshaped as wide. It is then saved to a temporary file which will contain a single imputed data set. This ends the main loop:

```
  qui gen iter = `iter' + `repoffset'

  keep pid iter state oldstate t

  reshape wide oldstate state, i(pid) j(t)

  tempfile mibo`iter'
  save `mibo`iter'',replace
}
```

The do-file finishes by assembling the temporary files from each pass through the main loop into a single file, saving to a filename distinguished by number:

```
use          `mibo1'
forvalues i = 2/$maxreps {
  append using `mibo`i''
}

save bsmi_it_`repoffset', replace
```

## 6.3 Program definitions

A large part of the code has been moved out of the main loop into two Stata programs, `do_imputations` and `do_imputations_it`, which carry out the internal and edge imputations, respectively. The main work done in these is to define the appropriate predictive variables, run the imputations, and assign the imputed values to the correct observations of the `state` variable.

### 6.3.1 `do_imputations` for internal gaps

Begin the program definition, with one argument, `i`, which is the index of the calling loop:

```
program define do_imputations
args i
```

Imputation of the internal gap alternates, imputing one end of the longest gap, then the other end of the next shortest gap, and so on. If the value of the `i` macro is an odd number, we impute the end of the gap, otherwise the beginning. We define the `last` and `next` variables accordingly: if we are imputing the last element of the longest gap, then `next` is `state` with a lead of 1, and `last` is `state` with a lag of `$longestgap`. When imputing at the start of a gap, then `last` is `state` lagged by 1, and `next` is `state` with a lead of `$longestgap + 1 - `i'`. As the gap length shortens the longer of lead or lag reduces by 1 each iteration. Note that we also pick up the lagged and lead values of any other variables used at this point (here `ol` and `on` from `obstype`). The local macro `seqno` indexes where in the gap the operation is focused, and will be used to help identify which observations to impute:

```
if mod(`i',2) == 1 {
  // i is odd: imputing at the end of the gap
  qui by pid: replace last = state[_n - $longestgap - 1 + `i']
  qui by pid: replace next = state[_n + 1]
  qui by pid: replace ol = obstype[_n - $longestgap - 1 + `i']
  qui by pid: replace on = obstype[_n + 1]
  local seqno = $longestgap + 1 - int((`i'+1)/2)
}
else {
  // i is even: imputing at the start of the gap
  qui by pid: replace last = state[_n - 1]
  qui by pid: replace next = state[_n + $longestgap + 1 - `i']
  qui by pid: replace ol = obstype[_n - 1]
  qui by pid: replace on = obstype[_n + $longestgap + 1 - `i']
  local seqno = int(`i'/2)
}
```

In the example, there is a variable `obstype` which should be interacted with `next` and `last`. However, this causes problems with convergence, so we collapse two categories. This is a specific requirement of the example data set, but it illustrates how one might deal with modelling problems. Such problems are likely to arise!

```
capture drop n2 l2
qui recode next 3=2, generate(n2)
qui recode last 3=2, generate(l2)
```

Set up the data set for imputation under Stata's `mi` framework:

```
mi set wide
mi register imputed state
```

Carry out the `mi` imputation. Note that this is the unique location where the main imputation model is defined, and if a different model is needed, the change must be made here. The `add(1)` option creates one imputation, `force` causes it to continue though some explanatory variables have missing values (a problem for conventional imputation but not so here), and `augment` causes it to use augmented multinomial logistic regression if perfect prediction is encountered. The `iterate(40)` option causes it to stop with an error if it exceeds

40 iterations, which is a good sign that it is never going to converge (numbers other than 40 might be appropriate). The `capture` prefix allows the imputation to fail without crashing the Stata job.

```
capture mi impute mlogit state i.next##c.t##c.t i.last##c.t##c.t ///
                              before1 before2 before3 ///
                              after1  after2  after3 ///
                              i.on##i.n2 i.ol##i.l2, ///
                 add(1) force augment noisily iterate(40)
```

If the imputation fails to converge, it will return an error code of 430. In this case, fit a much simpler model, one that enforces longitudinal consistency but not much more. If the imputation failed for any other reason, Stata will now crash with the appropriate error.

```
if (_rc==430) {
  di in red "NO CONVERGENCE, fitting minimal model"
  mi impute mlogit state i.next i.last, ///
    add(1) force augment noisily iterate(40)
}
else if _rc {
  exit _rc
}
```

The `mi` procedure will have created the variable `_1_state` with predicted values for every case. We want to use these values only for the gap element to which this pass through the loop corresponds. The variable `canassign` will be 1 for the appropriate cases, where `tw`, the month number within the gap, corresponds to the element we should be imputing (which in turn depends on the length of the current gap and the length of the longest gap permitted).

```
gen canassign = `seqno' - int(($longestgap - lg)/2) == tw
di "Putting imputed values in place in internal gap"
replace state = _1_state if missing(state) & canassign
```

We now drop unneeded variables, unset the `mi` state, and end the `do_imputations` program:

```
drop canassign
drop _*_state
capture drop state_*_

mi unset

end
```

#### 6.3.2 Predict initial and terminal gaps: `do_imputations_it`

The program to predict initial and terminal gaps takes two arguments, one distinguising initial vs terminal, the other the loop index:

```
program define do_imputations_it
args it i

if ("`it'"=="i") {
local gaptype "initial"
}
if ("`it'"=="t") {
local gaptype "terminal"
}
```

As in the internal gap imputation, we re-define the relevant variables, but here our lag or lead is always 1:

```
qui by pid: replace next = state[_n+1] if missing(next)
qui by pid: replace last = state[_n-1] if missing(last)
capture drop n2 l2
qui recode next 3=2, generate(n2)
qui recode last 3=2, generate(l2)
```

As before, we set up for `mi` and impute. The model used depends on which end we are imputing, but they are symmetric. Again we accommodate a failure to converge and provide a simpler alternative.

```
mi set wide
mi register imputed state

if ("`it'"=="i") {
  capture mi impute mlogit state i.next##c.t ///
                                  after1  after2  after3 ///
                                  i.obstype##i.n2, ///
                     add(1) force augment iterate(40)
  if (_rc==430) {
    di in red "NO CONVERGENCE, fitting minimal model"
    mi impute mlogit state i.next, ///
         add(1) force augment noisily iterate(40)
  }
}

if ("`it'"=="t") {
  capture mi impute mlogit state i.last##c.t ///
                                  before1 before2 before3 ///
                                  i.obstype##i.l2, ///
                     add(1) force augment iterate(40)
  if (_rc==430) {
    di in red "NO CONVERGENCE, fitting minimal model"
    mi impute mlogit state i.last, ///
         add(1) force augment noisily iterate(40)
  }
}
```

We can assign the imputation to cases where, for initial gaps, the month number `t` is equal to the actual gap length plus one decremented for each iteration, and for the terminal, the total length (73) minus the actual gap length incremented for each iteration:

```
if ("`it'"=="i") {
  gen canassign = t == igl + 1 - `i' & initgap & igl<=$longestgapit
}
else if ("`it'"=="t") {
  gen canassign = t == $totlen -tgl + `i' & termgap & tgl<=$longestgapit
}

replace state = _1_state if missing(state) & canassign
```

Finally, drop unneeded variables, unset `mi` and end.

```
drop canassign
drop _*_state
capture drop state_*_
mi unset

di "Finished `gaptype'"
end
```

## 6.4 `bsmicombit` to wrap it all up

Concatenate the multiple files from the parallel invocation of `bsmi_it.do`:

```
use bsmi_it_0.dta

append using bsmi_it_3.dta
append using bsmi_it_6.dta
append using bsmi_it_9.dta

sort pid iter
```

The `stripe` command from SADI creates a useful string representation of the sequence:

```
stripe state*, gen(s1)
stripe oldstate*, gen(s2)
```

For convenience, create a record with iteration number 0 that is the original data, by creating two versions of iteration 1, and copying the `oldstate` information into `state`:

```
expand 2 if iter==1
sort pid iter

replace iter = 0 if iter==1 & iter[_n+1]==1
```

```
replace s1 = s2 if iter==0
forvalues x = 1/73 {
  replace state`x' = oldstate`x' if iter==0
}
```

Using "regular-expression" text-matching functions makes it a little easier to differentiate between complete, imputed and incomplete cases, before saving the final file of imputations:

```
gen rectype = 0
by pid: replace rectype = 1 if !regexm(s1[1],"\.")
by pid: replace rectype = 2 if regexm(s1[1],"\.")
by pid: replace rectype = 3 if !regexm(s1[2],"\.") & rectype == 2
label define rectype 1 "Complete" 2 "Incomplete" 3 "Imputed"
label values rectype rectype

save bsmicombit, replace
```

Finally, to look at some of the results:

```
list iter s1 if rectype==3, sepby(pid)
```

# References

Halpin, B. (2012). *Multiple imputation for lifecourse sequence data* (Working Paper No. WP2012-01). Dept of Sociology, University of Limerick. Ireland. Retrieved from http://www.ul.ie/sociology/pubs/wp2012-01.pdf